Progress-Sensitive Security for SPARK

Willard Rafnsson[1]([✉]), Deepak Garg[2], and Andrei Sabelfeld[3]

[1] Carnegie Mellon University, Pittsburgh, USA
willardthor@cmu.edu
[2] Max Planck Institute for Software Systems, Kaiserslautern
and Saarbruecken, Germany
[3] Chalmers University of Technology, Gothenburg, Sweden

Abstract. SPARK 2014 is a safety critical language subset of Ada developed by Altran and used for developing safe and secure software by major industrial players in the aviation, commercial, medical, space, and military domains. This paper puts a spotlight on the SPARK flow analysis. Articulating the boundaries of what is achievable by the analysis, we spell out attacks to exploit termination, progress, resource exhaustion, and timing channels. We harden the analysis to achieve security against stronger attackers, with the focus on progress-sensitive security as our baseline. Instead of redesigning and reimplementing the enforcement, we leverage known flow analyses for weaker attackers by a transform on program dependence graphs. We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. A case study, derived from publicly available code for a control unit of a missile, indicates the usefulness of the approach.

1 Introduction

SPARK is a safety critical language subset of Ada developed by Altran and used by industry in the aviation, commercial, medical, space, and military domains. Applications range from programming jet engines (Lockheed Martin) to military aviation (EuroFighter), UK's air traffic control system (Altran), cross-domain guards (Rockwell Collins), smart card OS (MULTOS), biometrics software (NSA), and multi-level security systems (Secunet) [42].

SPARK 2014. A recent major overhaul of SPARK has led to SPARK 2014 [44], a language and accompanying tools for developing safe and secure software. To aid security verification, a flow analysis is integrated in the compiler to track information flow in SPARK programs and is used in applications like separation kernels [29] and multi-level workstations [39].

Information Flow Security. The security model of SPARK programs draws on information flow tracking. The goal is to track the propagation of data from sources (inputs) to sinks (output) as information is manipulated by programs. For systems whose sources and sinks are classified into secret and public (or more

complex classifications [18]), the baseline policy is *noninterference* [16,21] that prevents secret inputs from affecting public outputs.

There are different ways in which noninterference can be broken, corresponding to different information flow channels. An *explicit* flow results from a data flow from the right-hand side to the left-hand side in an assignment. An *implicit* [19] flow is via control flow: for example, branching on a secret and outputting different public values in the branches is an implicit flow that leaks information about the secret without any explicit leaks. The *termination channel* [48] is another source of potential leaks: a program that loops on a secret and outputs a public value on exiting the loop reveals whether the loop has terminated and therefore leaks information about the secret guard. A generalization of this channel is the *progress channel* [7] that can be used to leak information about secrets via the progress of public outputs. In contrast to the one-bit termination channel, this channel allows leaking secrets in their entirety by brute force attacks [7]. Other channels of interest are *resource exhaustion* and *timing* [37], which allow the attacker to learn secret information by observing abnormal behavior and time variation, respectively.

SPARK Security Examined. Usage of the SPARK flow analysis in industry is encouraging. It makes the following questions important. What attacks does it prevent? How can it be extended to achieve security against more powerful attackers? Can it lead to a general methodology applicable to similar analyses?

This paper puts a spotlight on the flow analysis in SPARK GPL 2015. Released April 28 2015, it is, as of January 1 2016, the latest GPL edition of SPARK 2014. To articulate the boundaries of what it can achieve, we demonstrate that the analysis successfully tracks explicit and implicit flows and spell out attacks to exploit termination, progress, resource exhaustion, and timing channels.

SPARK Security Improved. With the goal to harden the analysis against stronger attackers, we set our baseline at the progress-sensitive security policy [8,12,31,32]. This policy is a natural generalization of noninterference to programs with output, in contrast to its progress-insensitive counterpart [7,8,12] that needs to carve out leaks due to computation progress. Further, as mentioned earlier, ignoring the progress channel implies opening up brute force leaks that may extract secrets in their entirety. Our key goal is to design a general approach that allows leveraging existing analysis and tools for explicit and implicit flows, such as SPARK flow analysis to enforce the stronger progress-sensitive security. This goal is particularly important given the state of the art where the vast majority of the information flow tools in addition to SPARK (e.g. FlowFox [22], JSFlow [24] and IFC4BC [10] for JavaScript, Jif [30], Paragon [14] and JOANA [23] for Java, FlowCaml [40] for Caml, all discussed in Sect. 8) are currently only able to enforce progress-insensitive security.

Achieving Progress-Sensitive Security. With this main goal at hand, the core idea for enforcement is as follows. We set out to leverage two independent components: graph-based analysis for explicit/implicit flows and termination

analysis. There have been many successful efforts on developing such components, with the above-mentioned information flow tools for the former and much encouraging progress on the latter [17, 27, 45]. Facilitated by the latter, Moore et al. [28] show how to use *termination oracles* for termination-sensitive information flow analysis. Similarly, we parametrize our approach in the termination analysis to determine which loops terminate and perform a graph transform on the program dependence graph where we represent termination and progress flows by injecting additional edges going out of potentially diverging loops. This lets us reuse graph-based analyses, e.g. the one by Horwitz [26] that is behind the SPARK flow analysis, since we can simply apply it to the transformed graph. The elegance of this approach is that even if a trivial termination analysis ("all loops might diverge") is plugged into the framework, we get a sound and meaningful enforcement of progress-sensitive noninterference corresponding to Smith's and Boudol and Castellani's canonical restrictions for the termination channel [13, 41].

We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. We apply the source-to-source transformation on a case study with a control unit of a missile, loosely based on publicly available code by Hilton [25]. We formulate desired properties, such as "the orientation sensors may not affect self-destruction", in terms of information-flow policies and demonstrate how our enforcement verifies these properties.

Contributions. The paper's major contributions are (i) the attacks on the SPARK flow analysis to demarcate its boundaries, (ii) leveraging a progress-insensitive SPARK flow analysis (by changing the analyzer conservatively, or through source-to-source transformation) to enforce progress-sensitive noninterference, and (iii) a case study with a missile code controller to demonstrate the usefulness of the approach. While our work is motivated by improving the SPARK flow analysis, we believe the overall idea is portable to other approaches and tools. Thus, we present our results more generally. For example, our framework is graph-based, which opens up possibilities for natural adoption to other graph-based tools such as JOANA [23]. Combining the major and minor contributions, the paper contributes the following:

- Attacks illustrating the boundary of what SPARK's flow analysis can achieve, leaking via termination, progress, resource exhaustion, and timing (Sect. 2);
- A policy framework for expressing progress-(in)sensitive security conditions (Sect. 4) for an imperative language (Sect. 3) at the heart of SPARK;
- A general graph-based approach for dependency analysis using termination oracles to achieve progress-sensitive security (Sect. 5);
- A general graph-based framework for dependency analysis of reactive programs, also distinguishing output content from output presence (Sect. 5);
- Soundness of the graph-based enforcement for the core language (Sect. 5);
- Source-to-source transform leveraging existing graph-based flow analyses in a modular fashion (Sect. 6) to achieve progress-sensitive security; and

– Case study with a control unit of a missile that verifies desired security properties (Sect. 7).

Our code compiles with GNAT GPL 2015. Released April 28 2015, it is, as of January 1 2015, the latest GPL edition of the Ada 2012 compiler. Our code can be found online [49].

Scope. While resource exhaustion and timing channels are important, we leave their consideration and exploration of more sophisticated attacks on the SPARK security analysis for future work. Typically, attacks on these channels require more efforts from the attacker and result in attacks with lower bandwidth [37]. For similar reasons, we leave declassification [38] out of the scope of the present work. Although important and wished for by the SPARK developers [35], the flow analysis in SPARK is useful even without declassification, as indicated by its deployments by Secunet [29,39] and as highlighted by our case study.

2   Attacks

We begin by providing evidence that SPARK's flow analysis is termination-, progress-, and timing-insensitive. We do this by providing minimal example programs which pass analysis yet leak information. Since SPARK's flow analysis implementation has no proof of soundness, this helps us identify the property it is meant to enforce, and thus how to improve it.

All our examples share the same structure: a *Main* file that reads a byte from standard input, and invokes a procedure `Leak` on said byte.

```
0  procedure Leak (H : in out Byte)
     with Global  => (In_Out => Standard_Output),
          Depends => (H => H, Standard_Output => Standard_Output);
```

This specification states that `Leak` performs I/O on its parameter `H` and the global `Standard_Output`, and that output on `Standard_Output` *only* depends on `Standard_Output`. That last bit is the *flow policy* of `Leak`. Our attacks, which differ only in how they implement `Leak`, aim to violate this flow policy while passing analysis, by making output on `Standard_Output` depend on `H`.

The source code for our attacks is in the appendix. In this section, we focus on the two attacks most relevant to our technical contributions (termination and progress), and summarize the other attacks when closing the section.

Termination. A flow analysis is *termination sensitive* when it tracks whether a value can affect termination behavior. To gauge whether SPARK's flow analysis is termination sensitive, we design `Leak` on the left of Fig. 1 such that the *presence of output* on standard output depends on whether the program enters an infinite loop, which depends on `H`. *Main* passes analysis with this implemetation of `Leak`. However, invoking `Leak` on input values 1 and 2 produces different observable behavior: with input 1, we see '!' on the standard output; with input 2, *Main* diverges. Thus, SPARK's flow analysis is *termination insenstive*.

```
0  procedure Leak (H : in out Byte) is      0  procedure Leak (H : in out Byte) is
   begin                                        K : Byte := 0;
     H := H;                                  begin
                                                 H := H;
   -- if H is even: nontermination.              while True loop
5  -- else terminate with output "!".       5      Write (Standard_Output, K);
     if H mod 2 = 0 then                            if K >= H then
       while True loop                                while True loop
         H := H;                                         H := H;
       end loop;                                      end loop;
10   end if;                                 10     end if;
     Write (Standard_Output,                        K := K + 1;
           Character'Pos('!'));                   end loop;
   end Leak;                                    end Leak;
```

Fig. 1. Termination leaks (left) and progress leaks (right) in SPARK

Progress. A termination insensitive flow analysis permits one bit to leak
through termination observations. Programs that pass such an analysis can leak
much more when a value can affect the progress the program makes on producing
its intermediate output [7]. A flow analysis that tracks such flows is *progress sen-
sitive*. The right panel of Fig. 1 shows the brute force attack by Askarov et al. [7]
modelled in SPARK. Here, Leak outputs on standard output all characters in
their ASCII number order up to the character numbered H, and diverges, thus
leaking all of H. Again, this program passes SPARK's flow analysis, indicating
that the flow analysis is *progress insensitive*.

Summary. We studied SPARK's flow analysis under three additional attacks:

– *Resource Exhaustion:* We replace nontermination in the progress attack with
  abnormal termination (a stack overflow). We give two examples; one allocates
  an array too large to fit on the stack, the other creates infinitely many stack
  frames through infinite mutual recursion. Both pass analysis.
– *Timing:* A flow analysis is *timing sensitive* when it tracks whether a value can
  affect the time an effect occurs. We replace nontermination in the termination
  attack with a computation which takes considerable time (selection-sorting
  2^{16} bytes). The attack passes analysis.
– *Explicit and Implicit flows:* As a sanity check, we provide two implementations
  of Leak: one creates an explicit flow from H to Standard_Output, the other
  an implicit flow. The attacks do *not* pass analysis.

Since SPARK detected the explicit and implicit flows, but failed to detect our
other attacks, it appears that SPARK enforces progress-insensitive security. As
demonstrated above, whole secrets can leak through progress. In this paper,
we harden SPARK's flow analysis to detect progress leaks, to enforce progress
sensitive security. Addressing the other attacks is out of scope of this paper.

3   Programs and Policies

We explain our ideas and results using a simple while language with flow annotations, inputs, outputs, and arrays, which is a stripped down version of SPARK. For a formal semantics and illustrative examples, see the appendix.

Programs. The syntax for our language is given in Fig. 2. Let p range over programs, b over blocks, x over array names, e over expressions, n over integers, c over channels, and $\odot$ over (total) binary integer operators. Here, $x[e]$ denotes index e in array x. To model non-array variables, we write x as syntactic sugar for $x[0]$. Statement c `<-` e outputs integer e to channel c, and c `->` $x[e]$ inputs an integer on c and stores it in $x[e]$. The rest is standard.

$$
\begin{array}{llll}
p ::= \texttt{skip} & b ::= \texttt{skip} & \quad\Big\|\quad & \texttt{if } e \ \{p_1\} \ \{p_0\}; \ p \to p_1; \ p \\
\quad | \ b; \ p & \quad | \ x[e] := e & & \texttt{if } e \ \{p_1\} \ \{p_0\}; \ p \to p_0; \ p \\
& \quad | \ c \ \texttt{<-} \ e & & \texttt{while } e \ \{p_1\}; \ p \to p_1; \ \texttt{while } e \ \{p_1\}; \ p \\
e ::= n & \quad | \ c \ \texttt{->} \ x[e] & & \texttt{while } e \ \{p_1\}; \ p \to p \\
\quad | \ x[e] & \quad | \ \texttt{if } e \ \{p\} \ \{p\} & & \text{For } b \in \{\texttt{skip}, x[e] := e', c \ \texttt{<-} \ e, c \ \texttt{->} \ x[e]\}: \\
\quad | \ e \odot e & \quad | \ \texttt{while } e \ \{p\} & & \quad b; \ p \to p
\end{array}
$$

Fig. 2. Program syntax (left) and CFG (right)

Control Flow Graphs. A control flow graph (CFG) represents a program as a directed graph. The CFG of a program p is defined by $\to$ in Fig. 2; p' is a node iff $p \to^* p'$, and (p', p'') is an edge iff p' and p'' are nodes and $p' \to p''$. We distinguish two nodes in the CFG of program p: the START node p and the END node $\texttt{skip}$. START is defined as the root of the graph. END has no outgoing edges. Conventionally, CFG nodes are blocks, b. This representation is obtained by dropping p from nodes of the form b; p and replacing $\texttt{if } e \ \{\_\} \ \{\_\}$ and $\texttt{while } e \ \{\_\}$ nodes with $\texttt{branch } e$. See the appendix for an illustrative CFG.

Semantics. A program executes in a memory $m : \mathbb{X} \times \mathbb{N} \to \mathbb{Z}$, which provides a (mutable) binding for every location of every array (initially all set to 0 in the initial memory m_0), and an environment $\boldsymbol{e} : \mathbb{C} \to \mathbb{Z}^\omega$, which provides an infinite stream of input values on every channel. We use a small-step reduction relation $(\boldsymbol{e}, m, p) \overset{o}{\to} (\boldsymbol{e}', m', p')$. Here, $o := \bullet \mid {!cv}$ is the *output* of the reduction step; if $p = c \ \texttt{<-} \ e; \ p'$, then $o = {!cv}$ where v is the value e evaluates to; otherwise, $o = \bullet$. The full definition of $\overset{o}{\to}$ is shown in the appendix. Let $\bar{o} = o_1 \ldots o_n$ denote a finite sequence of outputs, and let $\overset{\bar{o}}{\to} = (\overset{\bullet}{\to})^* \overset{o_1}{\to} (\overset{\bullet}{\to})^* \ldots (\overset{\bullet}{\to})^* \overset{o_n}{\to} (\overset{\bullet}{\to})^*$.

Our environments are *total* [32], i.e., never block output, and always provide input on request. This is a natural fit for SPARK, as safety-critical systems typically perform nonblocking I/O (e.g. on files and POSIX shared memory using `read()` and `write()` from the Single UNIX Specification). The endpoints of channels thus, in general, form a collective store which can change independently of the program, and provide input that depends on past output. However,

Clark and Hunt [15] have shown that *when reasoning about security* of deterministic programs (as in our case), environments can be simplified to streams. We use this simplification here. Programs can be composed securely under these environments as long as their scheduler is secure and deterministic. For a more complete and general treatment of composition, see [32,33].

Flow Policies. A flow policy expresses permitted flows between input and output channels. We are interested in two kinds of dependencies: where input affects the *presence* (i.e. occurrence) resp. *content* (i.e. value) of an output. The syntax of our flow policy language is given in Fig. 3. Let f range over flow policy specifications, and d over dependencies. The syntax c `=>` c' (resp. c `->` c') means that content (resp. presence) of output on c is allowed to depend on input on c'. For instance, a flow policy stating that (only) the presence of output on StdErr (standard error) is allowed to depend on input on StdIn (standard input) can be written as StdErr `->` StdIn; null. Every flow policy f straightforwardly yields a pair of functions (π, κ) where $\pi(c)$ (resp. $\kappa(c)$) is the set of input channels on which the presence (resp. content) of output on c may depend. We lift these functions to sets of channels: $\pi(C) = \bigcup_{c \in C} \pi(c)$ and $\kappa(C) = \bigcup_{c \in C} \kappa(c)$.

$$f ::= \texttt{null} \qquad d ::= c \texttt{ => } c$$
$$\mid \ d\,;\,f \qquad\qquad \mid \ c \texttt{ -> } c$$

Fig. 3. Syntax of flow policies

4   Security Property

Consider a fixed policy (π, κ). Our attackers observe all outputs on some output channels. An attacker or *observer* $\omega = (\omega_\pi, \omega_\kappa)$ is a pair where ω_π (resp. ω_κ) is the set of channels on which the presence (resp. content) of outputs is observed. If an observer sees the content of outputs on a channel, it can certainly detect the presence of outputs on the channel, so we require $\omega_\kappa \subseteq \omega_\pi$. Two environments are equivalent to an observer ω if the environments agree on all input channels that may flow to outputs visible to ω.

Definition 1 (ω-equivalence of e). e *and* e' *are ω-equivalent,* $e \sim_\omega e'$, *iff* $\forall c \in \pi(\omega_\pi) \cup \kappa(\omega_\kappa) \centerdot e(c) = e'(c)$.

The observables in an output are defined as follows: $!cv\!\restriction_\omega = \ !cv$ if $c \in \omega_\kappa$, $!cd$ if $c \in \omega_\pi \setminus \omega_\kappa$, and $\bullet$ otherwise (here d is a default output, like null or 0). We remove the unobservables of a sequence of outputs $\bar{o}$ follows: $\epsilon\!\restriction_\omega = \epsilon$, $(o.\bar{o})\!\restriction_\omega = \bar{o}\!\restriction_\omega$ if $o\!\restriction_\omega = \bullet$, and $(o\!\restriction_\omega).(\bar{o}\!\restriction_\omega)$ otherwise.

Definition 2 (ω-equivalence of $\bar{o}$). $\bar{o}$ *and* $\bar{o}'$ *are ω-equivalent,* $\bar{o} \sim_\omega \bar{o}'$, *iff* $\bar{o}\!\restriction_\omega = \bar{o}'\!\restriction_\omega$.

Our security property, *progress-sensitive noninterference* (PSNI), requires that under observably equivalent environments, a program must be able to componentwise observably-equivalently match observable outputs in its behaviors [8,12,31,32]. For an example involving PSNI, see the appendix.

Definition 3 (Progress-sensitive Noninterference). *p satisfies* PSNI *iff*
$$\forall \omega, e, e' \mathbin{\raisebox{0.2ex}{.}} e \sim_\omega e' \implies \forall \bar{o} \mathbin{\raisebox{0.2ex}{.}} (e, m_0, p) \xrightarrow{\bar{o}} \implies \exists \bar{o}' \mathbin{\raisebox{0.2ex}{.}} (e', m_0, p) \xrightarrow{\bar{o}'} \wedge \ \bar{o} \sim_\omega \bar{o}'.$$

5   Enforcement

SPARK implements a dependency analysis on control flow graphs that prevents all explicit and implicit information leaks, but does not prevent leaks due to progress and termination. In this section, we explain how to augment such a dependency analysis with a loop termination oracle to enforce the stronger property progress-sensitive noninteference (PSNI, Definition 3). While loop termination oracles have been combined with type sytems to enforce PSNI in prior work (e.g., [28]), our technical development makes three novel contributions: (1) We use a graph-based analysis to enforce PSNI (2) Our dependency analysis handles reactive programs, and (3) Our dependency analysis accounts for the difference between output content and output presence. In the following, we describe our analysis for the core language from Sect. 3 and prove that it enforces PSNI. The core language captures all essential features of SPARK, so generalizing the analysis to all of SPARK should not be difficult.

Standard Data- and Control-Dependency Analysis. SPARK's flow analysis uses standard dependency analysis [20,26], which we review briefly. We say that a node b in a CFG reads array x if b contains x in at least one location other than `x[...] := ...`. Dually, b writes to array x if $b = (x[e] := e')$. Node b reads a channel c if $b = (c \ \text{->} \ \dots \ [\dots])$. Dependency analysis outputs all the nodes of the CFG on which a given node is *data dependent* or *control dependent*. Data dependence arises due to data flow. E.g., in `x = 1; y = 3; z = x + 2; a = z`, the statements `z = x + 2` and `a = z` are data dependent on the statement `x = 1`, but not on `y = 3`. Similarly, in the example of Fig. 1 (right), the statements on lines 5 and 6 (output and branch `K >= H`, respectively) are data dependent on the statement `K := K + 1` on line 11.

Definition 4 (Data Dependence). *A node b is data dependent on node b' in a CFG G, written $dd_G(b', b)$, if there is a path $b' \to^* b \in G$ and there is an array that b' writes and b reads, or there is a channel that both b and b' read.*

Note that the statement in b does not have to be an assignment; the definition implies a data dependence from `x = y` to `c <- x` in program `x = y; c <- x`. Also, as commonly assumed by flow analyses in prior work, e.g. Jif [30] and Paragon [14], our definition of data dependence is *flow-insensitive*. This means it ignores the effects of writes in nodes strictly between b' and b ; in program `x = y; x = 0; z = x`, node `z = x` is data dependent on the node `x = y` by our definition, even though `x` is overwritten by a constant between the nodes. (We use some lemmas from [1] in our proofs, but this difference does not impact those lemmas.) For clues on how to make this definition flow-sensitive, see [23].

Control dependence captures influence due to branches. In the program `if (x > 0) { y = 1 } else { y = 2 }; z = 1`, both the nodes `y = 1` and

`y = 2` are control dependent on the branch node `x > 0`. However, the node `z = 1` is *not* control dependent on `x > 0` because it executes irrespective of the outcome of the test `x > 0`. There are many different definitions of control dependence in literature (see [34] for a survey). We define here the most standard notion of control dependence, which suffices for our purposes. We say that node b *post-dominates* b' if every path from b' to END passes through b.

Definition 5 (Control Dependence [1]**).** *A node b is control dependent on node b' in a CFG G, written $cd_G(b', b)$, if the following hold: (1) Either $b = b'$ or b does not post-dominate b' in G, and (2) There is a nontrivial path $b_1 \to \ldots \to b_k \in G$ with $b_1 = b'$, $b_k = b$ such that for all $i \in 2 \ldots k-1$, b post-dominates b_i.*

For block-structured languages such as SPARK and the core calculus of Sect. 3, a node b is control dependent on node b' iff b is a branch or loop condition and b' lies within that branch or loop. However, control dependence is defined on arbitrary CFGs, even those without block structure (we exploit this generality later). Combining data- and control-dependency analysis, we define *dependence* as the reflexive-transitive closure of the data- and control-dependence relations. For example, in the program of Fig. 1 (right), the `while` loop on line 7 is dependent on the statement `K := K + 1` on line 11 because the condition `K >= H` on line 6 is data-dependent on line 11, and line 7 is control-dependent on line 6. The set of all nodes on which a node b depends is called b's backward slice.

Definition 6 (Dependence and Backward Slice). *The dependence relation dep_G for CFG G is defined as $(dd_G \cup cd_G)^*$. The backward slice of node b, $BS_G(b) = \{b' \mid dep_G(b', b)\}$, is the set of all nodes on which b is dependent.*

Information Flow Control Using Dependency Analysis. The dependence relation dep_G captures all explicit and implicit flows, and, hence, can be used for enforcement of information flow policies. There are well-known algorithms to compute dependencies and backward slices efficiently, e.g., [26]. This analysis is already implemented in SPARK. However, noninterference enforced this way is progress-*in*sensitive because the dependency analysis described above does not take into account nonterminating loops. For instance, the program of Fig. 1 (right) passes SPARK's dependency analysis, even though it leaks H to a progress-sensitive adversary who can observe K. Additionally, the method so far has been limited to sequential programs where the adversary makes only one observation at the end of the program. We explain how the method can be adapted to enforce progress-sensitive noninterference on reactive programs, additionally accounting separately for output content and output presence.

Progress-sensitive Dependence. A leak due to progress happens when an attacker-visible output is pre-empted due to the nontermination of a branch with a secret branch condition. Our simple insight is that such leaks can be detected by a dependence analysis if we ensure the following:

Requirement 1. An output that can be reached *after* the end of a branch is dependent on the branch point if some loop in the branch may diverge.

To implement Requirement 1, we use a static termination analysis, often called a termination oracle [17, 27, 45]. This oracle determines which loops in the program may diverge. We add an edge from every node in such a loop to the END node of the CFG. It is easy to check that the modified CFG satisfies Requirement 1 if the termination oracle is *sound*, i.e., it flags all loops that diverge on some input. A trivial, sound termination oracle marks every loop as potentially non-terminating. The use of this oracle in our analysis causes every program that contains an attacker-visible output after a loop with a secret loop condition to be marked as leaky, irrespective of whether or not the loop diverges, which may result in false positives. This corresponds exactly to termination-sensitive analyses developed by Smith [41] and Boudol and Castellani [13]. False positives can be reduced using a more precise termination oracle. For example, the program `while (h <> h) { }; l = 1` does not have a flow (via progress or otherwise) from the input variable `h` to the output variable `l`, but the trivial oracle above will cause this program to be marked leaky by the analysis. On the other hand, a slightly better oracle that uses symbolic analysis to infer that `(h <> h)` is always `false` will cause the program to be accepted. In general, fewer false positives in the termination oracle translate to fewer false positives in our dependence analysis. Consequently, we present our analysis *parametrically* in the termination oracle, leaving it to the specific implementation to decide how many resources to devote to the oracle (and, hence, how much precision to obtain).

Definition 7 (Termination Oracle). *A termination oracle T is a function that maps a CFG to a subset of the CFG's nodes. T is sound if for every CFG G and every node $b \in G$, $b \in T(G)$ if there is a memory m and environment e such that b appears infinitely often in the reduction sequence starting from the state (e, m, START).*

Definition 8 (Progress-sensitive Graph). *Given a control flow graph G and a termination oracle T, the progress-sensitive CFG $\mathrm{ps}_T(G)$ is defined by adding to G the edges $\{(b, \mathsf{END}) \mid b \in T(G)\}$.*

For the program of Fig. 1 (right), a sound termination oracle T will say that the `while` loop on line 7 is nonterminating and, hence, $\mathrm{ps}_T(G)$ will contain an edge from the branch condition of the loop to the end of the program. This makes the output statement on line 5 dependent on the branch condition `K >= H` and, hence, a dependency analysis will discover the progress leak in the program. Note that $\mathrm{ps}_T(G)$ may not correspond to any block-structured program.

Enforcing PSNI with Content and Presence Distinction. Our analysis takes as input a policy f and a program p. It works as follows. Let G be p's CFG. We compute the progress-sensitive CFG $G' = \mathrm{ps}_T(G)$. Then, for each node $b \in G'$ that outputs to some channel c, we compute the backward slice of b in G', and check that the policy relation κ allows a flow to c from any channel c' on which an input is made in the backward slice. This ensures that information flows to the *content* of messages on c only in accordance with the

policy. To account for flows due to *presence* of outputs on c, we compute a second backward slice from the same node b, but after erasing the payload of the output in b. We check that the policy relation π (not κ) allows a flow to c from any channel c' on which an input is made in this backward slice. Thus, by computing two backward slices per output node, we capture separate observations of content and presence. In the sequel, we assume a fixed policy $f = (\kappa, \pi)$.

Definition 9 (Enforcement of PSNI). *Let p be a program with CFG G. Let $G' = ps_T(G)$. We say p passes the PSNI enforcement, written $\mathsf{check}_T(p)$, if the following hold for any node b of the form c `<-` e in G':*

1. *If c' `->` $x[e'] \in BS_{G'}(b)$ then $c' \in \kappa(c)$.*
2. *If c' `->` $x[e'] \in BS_{\hat{G}'}(\hat{b})$ then $c' \in \pi(c)$, where $\hat{G}'$ is obtained by replacing b with $\hat{b} = c$ `<-` `d` in G'.*

Our main theorem is that the enforcement above is sound: If $\mathsf{check}_T(p)$, then p satisfies PSNI. We prove the theorem using bisimulations on backward slices [1]. Our proof is inspired by a related proof for enforcement of progress-insensitive noninterference in a sequential language [50]. In contrast to that proof, our proof captures progress-sensitive noninterference for a reactive language. Handling reactivity is quite involved: With multiple outputs, we have to argue that the *order* of observable outputs (at different program points) is independent of secret inputs. To do this, we construct a hypothetical slice that is the union of slices from all outputs visible to a given adversary. See the appendix for details.

Theorem 1 (Soundness of Enforcement). *If T is a sound termination oracle and $\mathsf{check}_T(p)$, then p satisfies PSNI.*

6   Source-to-Source Transform

The previous section describes a CFG transformation which ensures Requirement 1 – that any outputs after a potentially divergent branch depend on the branch's condition, which is used to enforce PSNI. In this section, we describe a source-to-source transform that also implies Requirement 1. The transform can be used to enforce PSNI using a standard, *unmodified* (and, hence, closed-source) dependency analysis of the kind that exists for SPARK.

The goal of our source-to-source transform, like the CFG transform, is to add a direct path from every potentially infinite loop to the end of the program. If the existing dependency analysis supports programmatic exceptions, then the transform is trivial: Just before every potentially infinite loop (identified by the oracle T), we add a statement to raise an unhandled exception, conditional on an unsatisfiable predicate. This has the effect of simulating an edge from the loop to the end of the program because the exception is not handled anywhere. It is quite easy to see that this has the same effect as the CFG transform. For example, the program in the right of Fig. 1 would be transformed to the program in the left of Fig. 4. Observe the new line 12 with the `raise` statement.

```
0  procedure Leak (H : in out Byte) is       0  procedure Leak (H : in out Byte) is
     E : exception; -- new exception             E : Byte := 0; -- E := 1 when an
     X : Byte := 0;                                              -- exception is raised
     K : Byte := 0;                               K : Byte := 0;
     O : File_Type := Standard_Output;            O : File_Type := Standard_Output;
5  begin                                       5  begin
     H := H;                                      if E = 0 then H := H; end if;
     if X = 1 then raise E; end if;               if E > 0 then E := 1; end if;
     while True loop                              while E = 0 and then True loop
                                                    E := E;
10     Write (O, K);                        10     if E = 0 then Write (O, K); end if;
       if K >= H then                              if E = 0 and then K >= H then
         if X = 1 then raise E; end if;              if E > 0 then E := 1; end if;
         while True loop                             while E = 0 and then True loop
                                                        E := E;
15         H := H;                          15         if E = 0 then H := H; end if;
         end loop;                                 end loop;
       end if;                                   end if;
       K := K + 1;                               if E = 0 then K := K + 1; end if;
     end loop;                                  end loop;
20 end Leak;                                20 end Leak;
```

Fig. 4. Source-to-source transformation of the program in the left of Fig. 1, using exceptions (left) and using an emulation of exceptions (right)

If the flow analysis, like SPARK, does not track flows through exceptions [1], then the source-to-source transform can emulate an exception by adding a new boolean variable, say E, initially set to 0. E is set to 1 where the exception is to be raised, and the program is transformed to check that E is still 0 before executing any statement or entering any branch of the original program. This ensures that once the exception is "raised" (E is set to 1), no statement from the original program executes and control propagates to the end of the program silently. This transform can be defined formally, but we only illustrate it for the progress leak in Fig. 1 in the right panel of Fig. 4. Observe that there is now a dependency between the branch condition K >= H and the output statement on line 10.

We note that to enforce PSNI, the dependency analysis should be applied directly to the output of either of the two transforms described above, without any intervening compiler optimizations. Such optimizations can negate the effects of our transforms. For instance, constant propagation followed by dead code elimination would remove the two raise statements on lines 7 and 12 in the left panel of Fig. 4 and, hence, also remove the control dependencies introduced deliberately by the transform.

7   Case Study

We demonstrate the usefulness of our approach on a nontrivial application by implementing a control system for a cruise missile (derived from publicly available code by Hilton [25]), and applying our approach on the code to prove desired

---

[1] The SPARK 2014 documentation states that SPARK programs are allowed to raise exceptions, but may not handle them. However, in our experiments with SPARK GPL 2015, we found that the flow analysis did not track flows through exceptions.

properties. The code steers the missile towards a target coordinate, and deto-
nates a nuclear warhead once within range, or self-destructs in the event that
a device fails. The code is intended to be an illustrative model native to the
domain of SPARK. The code makes several simplifications (e.g. the missile flies
in 2D space), and there are many safety- and mission-critical considerations for
more realistic missile control systems that we have not considered. For details
on such considerations, see Hilton [25]. We give an overview of our case study
(all our code is online [49]).

The missile has three sensors: a *failure detector*, which reports when a device
has failed; an *intertial navigation system*, which provides spatial orientation and
displacement readings (via accelerometers and a ring laser gyroscope) for naviga-
tion by dead reckoning; and a *clock*, used to calculate orientation and displace-
ment from accelerometer readings through integration. Using these readings,
the code controls three actuators: a *watchdog*, which, if not actuated at regu-
lar intervals, triggers self-destruction (to avoid unwanted consequences of device
or software failure); a *nuclear warhead*, which is detonated when the missile
reaches its target; and *steering*, consisting of aerodynamic fins which the code
actuates for trajectory corrections. Architecturally, our code draws inspiration
from an existing case study on implementing a controller for a water boiler [43,
Section 7]. The *Main* module consists of a sense-control-actuate loop, in which
it commands the sensor modules to read from their device, uses these readings
to compute values to control the actuators, and invokes the actuator modules to
actuate their device. The inter-module information flows are given in Fig. 5.
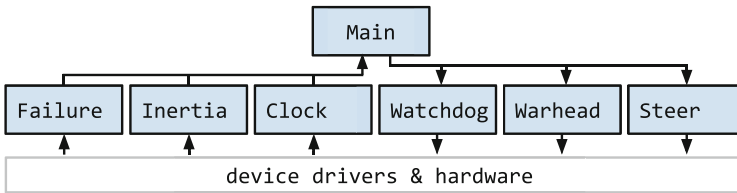
Fig. 5. Inter-module information flows in the missile control system

To illustrate our approach, we aim to prove that orientation does not affect
self-destruction. The body of the *Main* procedure, in the left of Fig. 6, is of
primary concern. Without our approach, since SPARK assumes loops terminate,
reaching "`Watchdog.Actuate`" in each iteration is deemed inevitable by SPARK,
so SPARK (incorrectly) claims the presence of a destruct event does not depend
on *any* input. However, if we instead apply the SPARK analysis on the code
resulting from applying our source-to-source analysis from Sect. 6 on the missile
control system source code, we get a different result: SPARK (correctly) infers
that the presence of a destruct event depends *only* on device failure. This can
be seen by inspecting the result of the transformation of the main loop, in the
right of Fig. 6. Since both loops are of the form "`while True loop`", any sound

termination oracle would flag them both as possibly diverging. Hence we emulate a raised exception before both loops, and add a check on variable `E` to each branch. SPARK no longer deems that reaching "`Watchdog.Actuate`" is inevitable; it now depends on the value of `E`. SPARK deems that the value of `E` depends on `Destruct`, since there is an assignment to `E` under a branch on `Destruct`. Since `Destruct` depends *only* on device failure, and since the only other assignment to `E` branches only on `E`, self-destruction depends only on device failure.

8   Related Work

We focus on the three most closely related areas of work: information flow tools, progress-sensitive security, and information flow analysis for SPARK.

Information Flow Tools. As mentioned before, much progress has been made on enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as FlowFox [22], JSFlow [24] and IFC4BC [10] for JavaScript, Jif [30], Paragon [14] and JOANA [23] for Java, FlowCaml [40] for Caml, LIO [47] for Haskell, and SPARK flow analysis [9] for SPARK. With the exception of the latest versions of LIO, these tools target *progress-insensitive noninterference* [7,8,12], allowing secrets to affect progress of public computation. With the focus on the termination and timing channels, Stefan et al. [46] introduced restrictions in LIO on side effects that follow secret branching, which help enforce stronger policies.

```
0
   while True loop
      -- [...] (sense, control)
      Steer.Actuate;
      if Destruct then
5         -- block watchdog.
         while True loop
            null;
         end loop;
      end if;
10    Watchdog.Actuate;
      if Detonate then
         Warhead.Actuate;
      end if; -- [...]
   end loop;
```

```
0  if E > 0 then E := 1; end if;
   while E = 0 loop
      E := E; -- [...] (sense, control)
      if E = 0 then Steer.Actuate; end if;
      if E = 0 and then Destruct then
5         if E > 0 then E := 1; end if;
         while E = 0 loop
            E := E; null;
         end loop;
      end if;
10    if E = 0 then Watchdog.Actuate; end if;
      if E = 0 and then Detonate then
         Warhead.Actuate;
      end if; -- [...]
   end loop;
```

Fig. 6. Main loop, before (left) and after (right) transformation

Progress-sensitive Security. *Progress-sensitive noninterference* [6,8,12,31,32] (PSNI) disallows progress leaks. PSNI is not susceptible to laundering secrets by brute-force attacks [7] or re-running programs [11]. A typical approach to enforcing PSNI is to disallow loops with secret guards, going back to Volpano and Smith's technique to deal with termination leaks [48], or to allow loops with secret guards but prohibit assignments to public variables that follow such loops [13,41]. While

the theory of progress-sensitive security has been explored [6,8,12,31,32], our work connects the theory with tools, showing how we can leverage a progress-insensitive tool (SPARK's flow analysis) to achieve PSNI. Related to our source-to-source transform, Russo et al. [36] discuss *magnification patterns* in the context of distinguishing flows in malicious and nonmalicious code. A magnification pattern in a control-flow graph consists of a branching on a secret guard inside of a loop. We note that in the absence of such patterns (as is sometimes the case in non-malicious code [36]), progress-sensitive security and progress-insensitive security coincide. Moore et al. [28] use *termination oracles* for termination-sensitive tracking. Their prototype implementation utilizes an SMT solver to analyze examples in a simple imperative language. While related, there are several distinguishing features of our work: we focus on practical information flow control in SPARK and push the approach to the full SPARK language; our case study goes beyond code snippets to a suite for a missile controller; on the theoretical side, our framework is graph-based, which opens up possibilities for natural adoption to other graph-based tools such as JOANA [23].

Information flow Analysis in SPARK. A line of work by Amtoft et al. shares with our work the motivation to improve SPARK's information flow analysis. Based on an expressive information logic [2], they enhance the information flow contract language to support compositional policies and conditional information flows [5]. They improve the precision of the analysis by breaking out of a limitation of the original analysis that treats arrays as indivisible entities and evaluate the approach on a collection of SPARK programs [4]. They extend the logical framework to produce machine-checkable formal certificates of correctness for verified code [3]. Extending the results by Amtoft et al. to guarantee progress-sensitive security is a promising direction for future work.

9   Conclusion

This paper puts a spotlight on the SPARK flow analysis. Articulating the boundaries of what is achievable by the analysis, we spell out the attacks to exploit such channels as termination, progress, resource exhaustion, and timing channels. We suggest how to harden the analysis to achieve security against stronger attackers, with the focus on progress-sensitive security as our baseline. Instead of redesigning and reimplementing the enforcement, we show how to leverage known flow analyses for weaker attackers by a transform on program dependence graphs. The graph transform represents termination and progress flows by injecting additional edges. We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. A case study with a control unit of a missile written in SPARK 2014 indicates the usefulness of the approach. Future work is focused on enriching the policy and enforcement mechanisms with possibilities for *declassification* [38], a feature on the wish list of the SPARK developers [35]. We are also interested in extending

the framework with treating resource exhaustion and timing leaks and exploring more sophisticated attacks.

References

1. Amtoft, T.: Slicing for modern program structures: A theory for eliminating irrelevant loops. Inf. Process. Lett. **106**(2), 45–51 (2008)
2. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL, pp. 91–102 (2006)
3. Amtoft, T., Dodds, J., Zhang, Z., Appel, A., Beringer, L., Hatcliff, J., Ou, X., Cousino, A.: A certificate infrastructure for machine-checked proofs of conditional information flow. In: Degano, P., Guttman, J.D. (eds.) Principles of Security and Trust. LNCS, vol. 7215, pp. 369–389. Springer, Heidelberg (2012)
4. Amtoft, T., Hatcliff, J., Rodríguez, E.: Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 43–63. Springer, Heidelberg (2010)
5. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, E., Hoag, J., Greve, D.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008)
6. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: CSF, July 2015
7. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008)
8. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proceeding of the IEEE Computer Security Foundations Symposium, July (2009)
9. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
10. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in webkit's javascript bytecode. In: Abadi, M., Kremer, S. (eds.) POST 2014 (ETAPS 2014). LNCS, vol. 8414, pp. 159–178. Springer, Heidelberg (2014)
11. Birgisson, A., Sabelfeld, A.: Multi-run security. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 372–391. Springer, Heidelberg (2011)
12. Bohannon, A., Pierce, B., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive noninterference. In: ACM Conference on Computer and Communications Security, pp. 79–90, November 2009
13. Boudol, G., Castellani, I.: Non-interference for concurrent programs and thread systems. Theor. Comput. Sci. **281**(1), 109–130 (2002)

14. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 217–232. Springer, Heidelberg (2013)
15. Clark, D., Hunt, S.: Noninterference for deterministic interactive programs. In: Workshop on Formal Aspects in Security and Trust (FAST 2008), October 2008
16. Cohen, E.S.: Information transmission in sequential programs. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, Academic Press, pp. 297–335 (1978)
17. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
18. Denning, D.E.: A lattice model of secure information flow. Comm. ACM **19**(5), 236–243 (1976)
19. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. ACM **20**(7), 504–513 (1977)
20. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
21. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 11–20, April 1982
22. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: A web browser with flexible and precise information flow control. In: ACM Conference on Computer and Communications Security (2012)
23. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009)
24. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: Proceeding of the 29th ACM Symposium on Applied Computing (2014)
25. Hilton, A.J.: High Integrity Hardware-Software Codesign. Ph.D. thesis, The Open University, April 2004
26. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, pp. 35–46 (1988)
27. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
28. Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: ACM Conference on Computer and Communications Security, pp. 881–893(2012)
29. The Muen Separation Kernel. http://muen.codelabs.ch/
30. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java Information Flow. Software release. Located at, July 2001 http://www.cs.cornell.edu/jif
31. O'Neill, K., Clarkson, M., Chong, S.: Information-flow security for interactive programs. In: Proceedings of the IEEE Computer Security Foundations Workshop, pp. 190–201, July 2006
32. Rafnsson, W., Hedin, D., Sabelfeld, A.: Securing interactive programs. In: Proceedings of the IEEE Computer Security Foundations Symposium, June 2012
33. Rafnsson, W., Sabelfeld, A.: Compositional security for interactive systems. In: CSF, pp. 277–292 (2014)
34. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**, 5 (2007)

35. Refined Information Flow Requirement. http://lists.forge.open-do.org/pipermail/spark2014-discuss/2012-December/000683.html
36. Russo, A., Sabelfeld, A., Li, K.: Implicit flows in malicious and nonmalicious code. 2009 Marktoberdorf Summer School (IOS Press) (2009)
37. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
38. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009)
39. Workstation, M.: High-Security Framework, Pilot, and Formalization Architecture. http://www.secunet.com/fileadmin//sina_downloads/Produktinfo_englisch/SINA-Multilevel_Brochure_en.pdf
40. Simonet, V.: The Flow Caml system. Software release. Located at, July 2003. http://cristal.inria.fr/~simonet/soft/flowcaml
41. Smith, G.: A new type system for secure information flow. In: Proceedings of the IEEE Computer Security Foundations Workshop, pp. 115–125, June 2001
42. SPARK (programming language). http://en.wikipedia.org/wiki/SPARK_%28programming_language%29
43. Development, T.: Support: INFORMED Design Method for SPARK. http://docs.adacore.com/sparkdocs-docs/Informed.htm
44. SPARK (2014). http://www.spark-2014.org/
45. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. ACM Trans. Program. Lang. Syst. **32**(3), Article no. 8, 70 (2010)
46. Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J.C., Maziéres, D.: Addressing covert termination and timing channels in concurrent information flow systems. In: ICFP, pp. 201–214 (2012)
47. Stefan, D., Russo, A., Mitchell, J., Mazières, D.: Flexible dynamic information flow control in haskell. In Proceedings of the Haskell Symposium, pp. 95–106. ACM (2011)
48. Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proceedings of the IEEE Computer Security Foundations Workshop, pp. 156–168, June 1997
49. Rafnsson, W., Garg, D., Sabelfeld, A.: Progress-Sensitive Security forSPARK. Full version: http://research.precise.li/pub/2016essos
50. Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based noninterference and its modular proof. In: PLAS, pp. 31–44 (2009)